CSE 332: Data Structures
Disjoint Set Union/Find

Richard Anderson, Steve Seitz
Winter 2014

Announcements

• Last week of the quarter – lots of deadlines
• Exam Monday

Disjoint Set ADT

• Data: set of pairwise disjoint sets.
• Required operations
  – Union – merge two sets to create their union
  – Find – determine which set an item appears in
• Each set has a unique name: one of its members (for convenience)
  – {3,5,7}, {4,2,8}, {9}, {1,6}

Application: Building Mazes

Algorithm

• S = set of sets of connected cells
  – Initialize to {{1}, {2}, ..., {n}}
• W = set of walls
  – Initialize to set of all walls {{1,2},{1,7}, ...}
• Maze = set of walls in maze (initially empty)

While there is more than one set in S
  Pick a random non-boundary wall (x,y) and remove from W
  u = Find(x);
  v = Find(y);
  if u ≠ v then
    Union(u,v)
  else
    Add wall (x,y) to Maze
  Add remaining members of W to Maze

Tree-based Approach

Each set is a tree
• Root of each tree is the set name.

• Represent: {3,5,7}, {4,2,8}, {9}, {1,6}
• Support: find(x), union(x,y)
Up-Tree for DS Union/Find

**Observation:** we will only traverse these trees upward from any given node to find the root.

**Idea:** reverse the pointers (make them point up from child to parent). The result is an up-tree.

Initial state

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Intermediate state

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Roots are the names of each set.

Find Operation

Find(x) follow x to the root and return the root.

Union Operation

Union(i, j) - assuming i and j roots, point i to j.

Simple Implementation

- Array of indices

<table>
<thead>
<tr>
<th>up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>up[x] = -1 means x is a root.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

| 1   |
| 2   |
| 3   |
| 4   |
| 5   |
| 6   |
| 7   |

<table>
<thead>
<tr>
<th>Find(1) n steps!!</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

Implementation

```c
void Union(int x, int y) {
    assert(up[x]<0 && up[y]<0);
    up[x] = y;
}
```

```c
int Find(int x) {
    while(up[x] >= 0) {
        x = up[x];
    }
    return x;
}
```

runtime for Union: runtime for Find:

Amortized complexity is no better.

A Bad Case

Union(1,2) Union(2,3) Union(n-1,n)
Two Big Improvements

Can we do better?  Yes!

1. Union-by-size
   • Improve Union so that Find only takes worst case time of $\Theta(\log n)$.

2. Path compression
   • Improve Find so that, with Union-by-size, Find takes amortized time of almost $\Theta(1)$.

Union-by-Size

Union-by-size
– Always point the smaller tree to the root of the larger tree

Example Again

Analysis of Union-by-Size

• Theorem: With union-by-size an up-tree of height $h$ has size at least $2^h$.
• Proof by induction
   – Base case: $h = 0$. The up-tree has one node, $2^0 = 1$
   – Inductive hypothesis: Assume true for $h-1$
   – Observation: tree gets taller only as a result of a union.

Analysis of Union-by-Size

• What is worst case complexity of Find($x$) in an up-tree forest of $n$ nodes?

• (Amortized complexity is no better.)
Worst Case for Union-by-Size

- n/2 Unions-by-size
- n/4 Unions-by-size

Example of Worst Cast (cont’)

After \( n - 1 = n/2 + n/4 + \ldots + 1 \) Unions-by-size

If there are \( n = 2^k \) nodes then the longest path from leaf to root has length \( k \).

Array Implementation

- Can store separate size array:
  
<table>
<thead>
<tr>
<th>up</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

Elegant Array Implementation

- Better, store sizes in the up array:
  
<table>
<thead>
<tr>
<th>up</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

Path Compression

- To improve the amortized complexity, we’ll introduce a new idea:
  - When going up the tree, improve nodes on the path!
  - On a Find operation point all the nodes on the search path directly to the root. This is called “path compression.”

Code for Union-by-Size

```c
S-Union(i, j) {
    // Collect sizes
    si = -up[i];
    sj = -up[j];

    // verify i and j are roots
    assert(si >=0 && sj >=0)
    // point smaller sized tree to
    // root of larger, update size
    if (si < sj) {
        up[i] = j;
        up[j] = -(si + sj);
    } else {
        up[j] = i;
        up[i] = -(si + sj);
    }
}
```
Self-Adjustment Works

Draw the result of Find(5):

Code for Path Compression Find

```c
PC-Find(i) {
    //find root
    j = i;
    while (up[j] >= 0) {
        j = up[j];
        root = j;
    }
    //compress path
    if (i != root) {
        parent = up[i];
        while (parent != root) {
            up[i] = root;
            i = parent;
            parent = up[parent];
        }
    }
    return(root)
}
```

Complexity of Union-by-Size + Path Compression

- Worst case time complexity for...
  - ...a single Union-by-size is:
  - ...a single PC-Find is:

- Time complexity for \( m \geq n \) operations on \( n \) elements has been shown to be \( O(m \log^* n) \).
  [See Weiss for proof.]
- Amortized complexity is then \( O(\log^* n) \)
- What is \( \log^* \) ?

The Tight Bound

In fact, Tarjan showed the time complexity for \( m \geq n \) operations on \( n \) elements is:

\[
\Theta(m \alpha(m, n))
\]

Amortized complexity is then \( \Theta(\alpha(m, n)) \).

What is \( \alpha(m, n) \)?

- Inverse of Ackermann’s function.
- For reasonable values of \( m, n \), grows even slower than \( \log^* n \). So, it’s even “more constant.”

Proof is beyond scope of this class. A simple algorithm can lead to incredibly hardcore analysis!